
**VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ**
**BRNO UNIVERSITY OF TECHNOLOGY**

**FAKULTA INFORMAČNÍCH TECHNOLOGIÍ**
**FACULTY OF INFORMATION TECHNOLOGY**

**ÚSTAV INTELIGENTNÍCH SYSTÉMŮ**
**DEPARTMENT OF INTELLIGENT SYSTEMS**


# TESTOVACÍ PROSTŘEDÍ PRO SPOUŠTĚNÍ PARALELNÍCH ÚLOH

**TESTING ENVIRONMENT VIA CLUSTER SOLUTION**

**BAKALÁŘSKÁ PRÁCE**
**BACHELOR'S THESIS**


**AUTOR PRÁCE** **PETER HOSTAČNÝ**
**AUTHOR**

**VEDOUCÍ PRÁCE** **Ing. ALEŠ SMRČKA, Ph.D.**
**SUPERVISOR**



**BRNO 2018**

**Vysoké učení technické v Brně - Fakulta informačních technologií**

Ústav inteligentních systémů Akademický rok 2017/2018

# Zadání bakalářské práce

Řešitel: **Hostačný Peter**
Obor: Informační technologie
Téma: **Testovací prostředí pro spouštění paralelních úloh**
**Testing Environment via Cluster Solution**
Kategorie: Operační systémy

Pokyny:

1. Nastudujte technologii virtualizace prostředí aplikací a strojů. Nastudujte problematiku spouštění paralelních úloh. Seznamte se se současným řešením paralelních výpočtů běžící v prostředí cluster.
2. Analyzujte požadavky pro řízení paralelních úloh v malém shluku počítačů (cluster) s různými parametry připojení. Navrhněte řešení pro nahrávání a spouštění paralelních úloh (tzv. cluster provisioning).
3. Implementujte systém pro spouštění paralelních úloh v rámci shluku počítačů. Systém bude umožňovat automatickou přípravu prostředí na vybraných počítačích, řízení zdrojů, sběr výsledků a případně rozložení zátěže podle předdefinovaných pravidel.
4. Ověřte správnost řešení na automatizované testovací sadě. Proveďte experimenty pro zjištění výkonnosti daného řešení.

Literatura:


- Gentzsch, W.: Sun Grid Engine: Towards Creating a Compute Power Grid, In Proc. of Cluster Computing and the Grid, IEEE/ACM, 200, doi: 10.1109/CCGRID.2001.923173
- Technologie Docker, URL: https://www.docker.com/
- Domovské stránky Open Grid Scheduler, URL: http://gridscheduler.sourceforge.net/


Podrobné závazné pokyny pro vypracování bakalářské práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

Technická zpráva bakalářské práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap (20 až 30% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním nepřepisovatelném paměťovém médiu (CD-R, DVD-R, apod.), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Smrčka Aleš, Ing., Ph.D.**, UITS FIT VUT
Datum zadání: 1. listopadu 2017
Datum odevzdání: 16. května 2018

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav inteligentních systémů
612 66 Brno, Božetěchova 2

doc. Dr. Ing. Petr Hanáček
*vedoucí ústavu*

# Abstrakt

Testos je projekt, ktorý sa zameriava na vytvorenie sady nástrojov podporujúcich automatizované testovanie softwaru. Jednou z jeho častí je výpočetný cluster, ktorého hlavným cieľom je počítanie paralelných úloh a poskytovanie výpočetných zdrojov ostatným nástrojom v projekte. Táto práca podrobne opisuje proces jeho návrhu, implementácie a integrácie s projektom Testos. Očakáva sa, že výsledný systém bude využívaný po dokončení projektu zbernice Testos, ktorá je k jeho integrácii potrebná.



# Abstract

Testos is a project that aims to create a set of tools supporting automated software testing. One of its components is computational cluster whose main goal is to execute parallel tasks and provide computational resources to other tools in the project. This work details the process of its design, implementation and integration with the Testos project. It is expected that the resulting system will be in use after Testos bus is completed.



# Kľúčové slová

Cluster, virtualizácia, paralelné úlohy, správa úloh, linuxové kontajnery, Docker, Testos

# Keywords

Cluster, virtualization, paralel tasks, task management, linux containers, Docker, Testos


# Citácia

HOSTAČNÝ, Peter. *Testovací prostředí pro spouštění paralelních úloh.* Brno, 2018. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Aleš Smrčka, Ph.D.

# Testovací prostředí pro spouštění paralelních úloh

## Prehlásenie

Prehlasujem, že som túto bakalársku prácu vypracoval samostatne pod vedením pána Ing. Aleša Smrčku, Ph.D. Uviedl som všetky literárne pramene a publikácie, z ktorých som čerpal.

.....
Peter Hostačný
17. mája 2018


## Poďakovanie

Týmto by som sa chcel poďakovať vedúcemu práce Ing. Alešovi Smrčkovi, Ph.D. za jeho odbornú pomoc a nasmerovanie na správne riešenie.

# Obsah

# Kapitola 1

# Úvod

Testos (Test Tool Set) [16] je projekt, ktorého hlavným cieľom je vytvorenie sady nástrojov podporujúcich automatizované testovanie softwaru. Nástroje v platforme Testos (viď. obrázok 1.1) kombinujú rôzne úrovne testovania a dajú sa rozdeliť do niekoľkých kategorií: testovanie založené na modeloch (Model-based), testovanie založené na požiadavkoch (Requirement-based), testovanie grafického uživateľského rozhrania (GUI), testovanie založené na dátach (Data-based) a dynamická analýza (Execution-based).

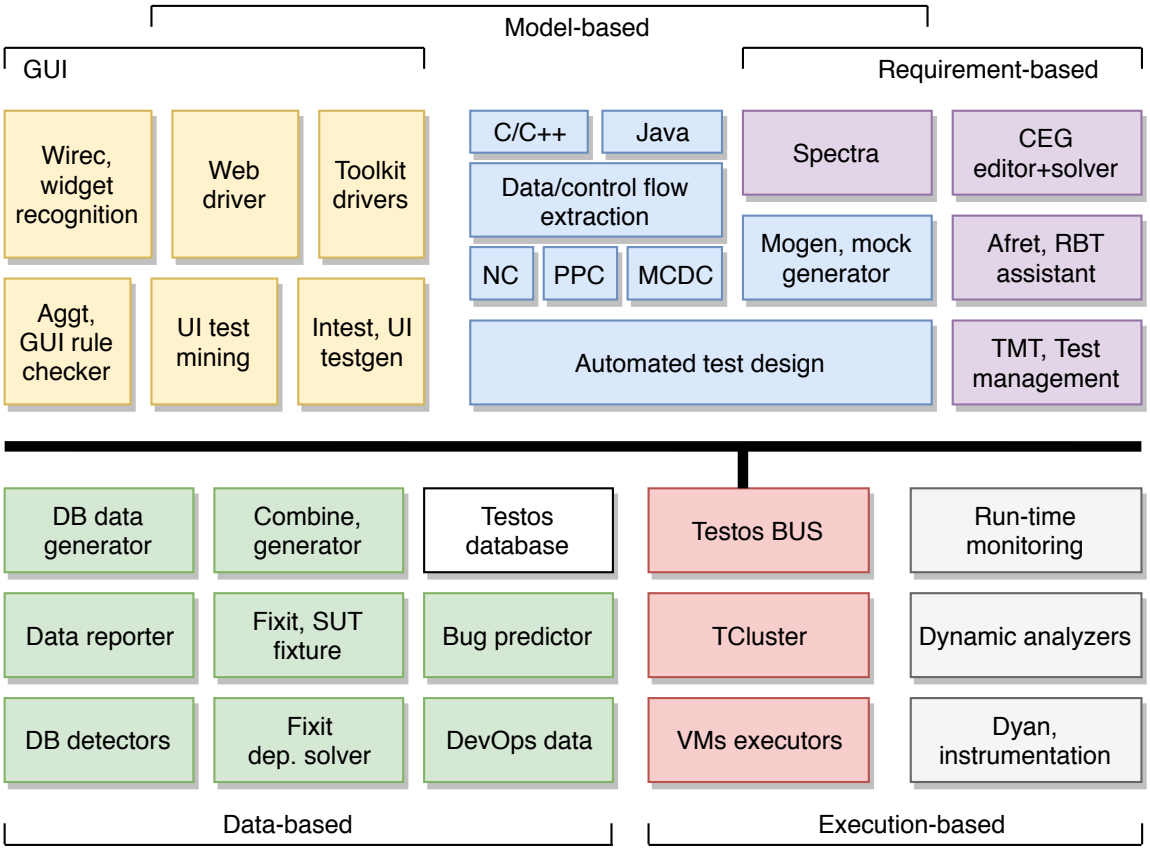
V aktuálnom vývoji nástrojov pre dynamickú analýzu programov (v rámci projektu Testos) sú nástroje pre analýzu využitia volania funkcií z knižníc v binárnych programoch [11], nástroje pre sledovanie paralelných kontraktov v jazyku Java [9], nástroj pre tvorbu kontajnerov pre stálosť testov [14], nástroj pre správu výpočetného clustru a zbernica podporujúca podsystémy platformy Testos [2].

Z vyššie vymenovaných nástrojov táto práca navrhuje a implementuje správu výpočetného clustru. Hlavným dôvodom k implementácii tohto clustru bola potreba spúšťať paralelné úlohy a poskytovať výpočetné zdroje k dispozícii ostatným nástrojom v projekte Testos. Počíta sa s tým, že výpočetné stroje môžu mať rôzne hardwarové a softwarové vybavenie a takisto, že počítanie úloh clustru nemusí byť ich primárny účel. Z toho dôvodu bolo potreba uvažovať nad použitím jednej z virtualizačných technológii. Vďaka virtualizácii sa dá zabezpečiť, že úloha sa bude vždy spúšťať v rovnakom prostredí, a takisto že úlohy nebudú môcť zasahovať do operačného a súborového systému výpočetných strojov. Keďže existuje viacero technológií virtualizácie, bolo potreba sa rozhodnúť, ktorá z nich sa najviac hodí pre účely clustru v projekte Testos. Porovnanie týchto technológií a zdôvodnenie výberu sa nachádza v kapitole 2.

Zoznam všetkých požiadavkov na výsledný systém spolu s jeho detailným návrhom je obsiahnutý v kapitole 3. Informácie týkajúce sa implementácie a testovania projektu sú v kapitole 4.

Očakáva sa, že zadávateľom úloh spúšťaných v tomto clustri budú ostatné nástroje projektu Testos a že tieto úlohy budú zadávané cez zbernicu projektu. Vzhľadom na to, že veľký počet týchto nástrojov je alebo bude písaných v jazyku Python 3, súčasťou implementácie je aj aplikačné rozhranie v tomto jazyku, ktoré komunikáciu po zbernici abstrahuje. Toto klientské rozhranie zjednodušuje vytváranie úloh, ich odosielanie clustru a preberanie výsledkov po ich ukončení. Klientské rozhrania sú bližšie popísané v podkapitole 4.3.

Ďalej v práci nasleduje záver, v ktorom sa nachádza zhodnotenie výsledku práce a návrhy na ďalšie pokračovanie projektu (kap. 5), prílohy obsahujúce zoznam typov správ posielaných po zbernici Testos (príloha A) a konfiguračné súbory aplikácií (príloha B).




Obr. 1.1: Štruktúra projektu Testos.

Zdroj: http://testos.org/

# Kapitola 2

# Virtualizácia a paralelné úlohy

**Paralelné úlohy** Existujú úlohy, ktoré potrebujú veľké množstvo výpočetných zdrojov a tým často aj dlhší čas potrebný k ich behu. Ak je možné takúto náročnú úlohu rozdeliť na niekoľko na sebe nezávislých podúloh, môžu sa tieto časti spúšťať paralelne a tým urýchliť čas potrebný k získaniu výsledku úlohy. K behu takýchto paralelných úloh sa potom dá využiť viacero nezávislých výpočetných strojov a výsledky na konci agregovať. K tomuto sa dnes používajú technológie, medzi ktoré patria:

- *cloud computing* – pod pojmom "cloud" rozumieme paralelný, distribuovaný počítačový systém pozostávajúci z viacerých prepojených virtuálnych strojov, ktoré sú vytvárané dynamicky podľa potreby užívateľov [5]. "Cloud computing" je potom spôsob využitia služieb, ktoré takýto systém ponúka.

- *grid computing* – tento termín bol pevne ustanovený v publikácii *The Grid: Blueprint for a New Computing Infrastructure* [6]. Označuje spôsob výpočtov v systéme, ktorý spája a využíva stroje rôznych parametrov, nachádzajúcich sa spravidla na rôznych geograficky rozptýlených miestach, prepojených počítačovou sieťou. Užívatelia v tomto systéme získavajú prístup k výpočetným zdrojom, ktorých fyzickú lokalitu zvyčajne nepoznajú alebo o nej majú iba málo informácií. Systém je teda podobný elektrickej sieti (angl. *grid*), v ktorej odberateľ dostane prístup k elektrickej energii, pričom jej pôvod mu nemusí byť známy [10].

- *cluster computing* – je typ výpočtov, ktoré využívajú viacero úzko prepojených výpočetných uzlov. Tieto uzly bývajú spravidla prepojené pomocou lokálnej počítačovej siete, bežia na rovnakom hardware a majú rovnaké softwarové vybavenie. Ich cieľom zvyčajne býva zvýšenie výkonu alebo dostupnosti určitých služieb, čo sa dá pomocou clustru docieliť za menšiu cenu ako behom jedného počítača s podobným výkonom a dostupnosťou [7].

Z predchádzajúcich popisov technológií vyplýva, že systém implementovaný v tejto práci sa kvôli rôznorodosti výpočetných uzlov viac podobá na výpočetný grid ako cluster. Keďže takáto rôznorodosť uzlov v tejto práci nie je želaná, na jej elimináciu sa používa virtualizácia. Vďaka použitiu virtualizácie je tento detail systému z pohľadu klienta odtienený a teda úlohy bežia v jednotnom prostredí. V práci sa naďalej bude používať termín cluster ako označenie systému, ktorý táto práca navrhuje a implementuje.

**Existujúce riešenia** Jedny z najznámejších systémov typu grid sú BOINC[1] a Sun Grid Engine[2]. Hlavné využitie projektu BOINC je vo veľkej škálovateľnosti a používa sa hlavne na zapojenie počítačov dobrovoľníkov z celého sveta pre vedecké výpočty v čase, keď by ich procesory boli inak nečinné. Vývoj projektu Sun Grid Engine bol zastavený a projekty.

Oba projekty by bolo zložité integrovať s projektom Testos takým spôsobom, aby komunikácia prebiehala po zbernici Testos.

**Virtualizácia** Virtualizácia vznikla na pôde firmy IBM v roku 1964, pričom v tom čase bol jej vývoj ešte veľmi pomalý. Po tom, ako bol v roku 1999 na trh uvedený komerčný software pre virtualizáciu platformy X86 firmou VMware, začalo obdobie jej prudkého vývoja a dnešné dni sa v informatike stretávame s virtualizáciou prakticky na každom kroku [12]. Poskytuje nesporné výhody, vďaka ktorým môžme (okrem iného) spúšťať aplikácie vo virtuálnom prostredí bez obáv o bezpčenosť hostiteľského operačného systému. Ďalšou výhodou virtualizácie je jednotnosť prostredia, v ktorom aplikácie bežia – to znamená, že ak si užívateľ beh aplikácie najskôr vyskúša v konkrétnom virtuálnom prostredí (napr. v linxovom kontajneri obsahujúcom distribúciu Debian), je veľmi pravdepodobné, že rovnakú aplikáciu bez problémov spustí aj na inom fyzickom stroji, za predpokladu, že použije rovnaké virtuálne prostredie (v tomto prípade rovnaký kontajner). Vďaka tejto vlastnosti je v clustri možné spúšťať paralelné úlohy na výpočetných uzloch s rôznym hardwarom (popr. softwarom) a očakávať, že sa pri použití rovnakého virtuálneho prostredia budú správať rovnako a takisto, že toto správanie bude reprodukovateľné.

V podkapitole 2.1 sa nachádza vysvetlenie a porovnanie dvoch technológii virtualizácie – linuxových kontajnerov a virtuálnych strojov.

## 2.1 Virtuálne stroje vs linuxové kontajnery

### Výhody použitia linuxových kontajnerov vo výpočetnom clustri

Hlavným dôvodom pre použitie linuxových kontajnerov oproti virtuálnym strojom je obrovská rýchlosť pri ich vytváraní a spúšťaní a takisto malá náročnosť na zdroje hostiteľského systému, potrebných na ich réžiu.

Pôvod tejto svižnosti kontajnerov je v tom, že jednotlivé kontajnery na rozdiel od virtuálnych strojov nepotrebujú plnú kópiu celého operačného systému spolu s jadrom a so všetkými knižnicami a takisto preto, že pri ich štarte nedochádza k procesu zavádzaniu systému [13].

Efektivita správy zdrojov pri použití kontajnerov vyplýva z faktu, že kontajnery tieto zdroje zdieľajú s hostiteľským operačným systémom a takisto aj medzi sebou navzájom. Je to z dôvodu behu iba jedného jadra operačného systému (u hostiteľa), ktoré ma správu zdrojov na starosti. Z toho dôvodu výpočetné zdroje na strojoch nie sú rezervované (a teda blokované), až do momentu, keď si o ne proces v kontajneri požiada [13].

Keďže sa v tejto práci počíta s tým, že hlavným cieľom výpočetných uzlov v clustri častokrát nebude počítanie úloh clustru, bola táto efektivita využitia zdrojov jedným z dôležitých faktorov pri výbere technológie.

[1] https://boinc.berkeley.edu/
[2] https://en.wikipedia.org/wiki/Oracle_Grid_Engine

### Nevýhody linuxových kontajnerov a ich využitia vo výpočetnom clustri miesto virtuálnych strojov

Ako už bolo spomenuté, narozdiel od virtuálnych strojov, bežiace kontajnery zdieľajú jadro s hostiteľským operačným systémom, čo znamená, že kontajnery vždy bežia s rovnakým jadrom operačného systému ako ich hostiteľ [13]. Využitie kontajnerov vo výpočetnom clustri so sebou teda prináša riziko, ktoré spôsobuje, že úlohy môžu na niektorých výpočetných strojoch bežať bez problémov a na iných zlyhať. To môže nastať ak úloha využíva určitú funkcionalitu jadra, ktorá sa v inom jadre (popr. inej verzii jadra) nenachádza alebo v ňom obsahuje chybu, popr. sa správa spôsobom, ktorý užívateľ neočakával. Keďže užívateľ v clustroch bežne nedefinuje, na ktorom z výpočetných strojov má presne úloha bežať, toto chovanie by mohlo spôsobiť nedeterministické chovanie a úloha by mohla byť ovplyvnená tým, na ktorom stroji by bola spustená.

Ak by sa tento problém vyskytoval v praxi príliš často, jedno z možných riešení by bolo presunutie aplikácie obsluhujúcej výpočetný stroj do virtuálneho stroja, ktorý by bol u všetkých výpočetných strojoch rovnaký. Toto by ale viedlo k strate výhody efektivity správy zdrojov, ktoré použitie kontajnerov prináša. Na druhej strane kontajnery by sa vždy vytvárali v rovnakom prostredí (v prostredí virtuálneho stroja) s vopred známym jadrom operačného systému a teda problémy nedeterminizmu pri behu úloh by sa vyriešili.

Druhým riešením vyššie uvedeného problému by bola identifikácia chýb spôsobených rôznymi jadrami operačných systémov a vylúčeniu problematických výpočetných strojov pri počítaní konkrétnych úloh. Druhé riešenie je v aktuálnej implemntácii clustru možné pomocou filtrovania výpočetných strojov a požiadavkov úloh na tieto stroje, podľa ktorých toto filtrovanie prebieha.

Ďalšou z nevýhod použitia kontajnerov je bezpečnosť. Vďaka tomu, že kontajnery zdieľajú jadro operačného systému s hostiteľom, akýkoľvek problém alebo zraniteľnosť jadra, ktorú aplikácia v kontajneri využije, zasiahne hostiteľský systém a tým pádom aj ostatné bežiace kontajnery [13]. Problémov týkajúcich sa bezpečnosti je v kontajneroch viacero ale keďže táto práca zatiaľ nepočíta s behom nedôveryhodných aplikácii, je bezpečnosť kontajnerov na dostatočnej úrovni aby tento aspekt nehral veľkú rolu pri výbere vhodnej technológie virtualizácie.

### Zhrnutie vlastností kontajnerov a virtuálnych strojov

Zhrnutie vlastností z predchádzajúcich podkapitol:

- Obe technológie — virtuálne stroje aj linuxové kontajnery — sa používajú pre izolovanie bežiacich aplikácii od ostatných aplikácii bežiacich na hostiteľskom systéme. Virtuálne stroje sú v tomto ohľade mierne bezpčenejšie ako kontajnery ale cenou je flexibilita, ktorú kontajnery ponúkajú.
- Vytváranie a spúšťanie kontajnerov je oproti virtuálnym strojom neporovnateľne rýchlejšie.
- Vytváranie kontajnerov je menej náročné na potrebný diskový priestor.
- Kontajnery poskytujú väčšiu efektivitu využívania výpočetných zdrojov.

- Pri použití hostiteľov s rôznym hardwarom a operačným systémom sa niektoré rozdiely premietnú aj do kontajnerov, čo môže ovplyvniť aplikácie spúšťané v kontajneroch naprieč clustrom.

Keďže je pre nás vyššia efektivita a rýchlosť pri spúšťaní úloh dôležitejšia ako maximálne zabezpečenie výpočetných strojov, rozhodli sme sa pre využitie technológie linuxových kontajnerov. Cluster by zároveň malo byť jednoduché integrovať s ďalšími technológiami a tým rozšíriť o nové možnosti spúšťanie úloh.

# Kapitola 3

# Analýza požiadavkov a návrh riešenia clustru

Táto kapitola analyzuje požiadavky na výsledný systém a dokumentuje jeho návrh. Návrh je rozdelený do niekoľkých podkapitol, pričom každá rozoberá detailne inú súčasť clustru. Cluster ako celok je popísaný v podkapitole 3.2, ktorá zároveň slúži ako rozcestník k jednotlivým častiam návrhu.

## 3.1 Požiadavky na systém

Zo zadania a z pravidelných schôdzok v rámci projektu Testos vzniklo viacero požiadavkov na výsledný systém. V tabuľke 3.1 sa nachádza ich zoznam a odkazy k riešeniu v tejto práci.

Existuje niekoľko vecí, ktoré cluster zatiaľ nepodporuje. Napríklad:

- *Autentizáciu/autorizáciu* – cluster je aktuálne navrhnutý ako otvorený systém kde majú všetci užívatelia pripojení na zbernicu projektu Testos oprávnenia na všetky úkony. Pre implementáciu autentizácie/autorizácie bude treba rozhodnúť, na ktorej vrstve a akým spôbom bude riešená, na čo sa táto práca nezameriavala.
- *Agregáciu výsledkov* – zatiaľ nebolo potreba výsledky nijak agregovať.
- *Správu úloh* – klient má v aktuálnej implementácii možnosť rušiť úlohy (bežiace aj čakajúce vo fronte). Okrem rušenia úloh existuje niekoľko ďalších úkonov, ktoré nie sú implementované, ako napríklad možnosť vypísať všetky aktuálne bežiace úlohy, pozastavenie úlohy alebo presun úlohy medzi výpočetnými uzlami. Kompletná správa clustru/úloh je mimo rozsah tejto práce ale počíta sa s tým, že v budúcnosti bude implementovaná.
- *Sprístupnenie grafického procesora (GPU) v kontajneroch* – docker s použitím predvolených nastavení nesprístupňuje žiadne zariadenia (angl. *devices*) v kontajneroch [1][3]. Pre využitie grafického procesora na výpočetnom stroji by teda bolo potreba vyriešiť niekoľko netriviálnych problémov ako napríklad jeho sprístupnenie v kontajneri, zdieľanie jeho ovládača a užívateľských knižníc medzi kontajnerom a hostiteľským systémom. Na schdôdzi projektu Testos sme usúdili, že je to mimo rozsah tejto práce.
- *Spúšťanie úloh na virtuálnych strojoch.*

| id | požiadavok | riešenie v podkapitolách |
|---|---|---|
| R1 | Systém umožňuje klientom zadávať nové úlohy. | 4.3, A.1.1 |
| R2 | Úlohy je možné parametrizovať. | 3.2.1, 4.3.2 |
| R3 | Klient má možnosť zrušiť vytvorené úlohy bez ohľadu na to, či už bežia. | 3.2.1, 4.3, A.1.2 |
| R4 | Cluster umožňuje klientom zisťovať stav úloh a prevziať si ich výsledky – informácie o behu, štandardný výstup (stdout), štandardný chybový výstup (stderr) a výstupné artefakty (súbory ukladané po ukončení behu úlohy). | 4.3, A.1.4, A.1.5, A.1.6, A.1.7 |
| R5 | Výsledky behu úloh sú ukladané v perzistentnom úložisku. | 3.2.4 |
| R6 | Cluster umožňuje pripájanie a odpájanie nových výpočetných uzlov za behu. | 3.2.3, A.4.1, A.4.2 |
| R7 | Komunikácia medzi jednotlivými počítačmi, ktoré cluster tvoria a využívajú, prebieha výlučne na zbernici projektu Testos. | 3.2.5 |
| R8 | Klient má k dispozícii aplikačné rozhranie Python, ktoré môže použiť na komunikáciu s výpočetným clustrom. | 4.3.2 |
| R9 | Cluster plánuje vykonávanie úloh s ohľadom na zdroje dostupné na výpočetných uzloch. | 3.2.2 |
| R10 | Úlohy sa spúšťajú vo virtualizovanom prostredí, ktorého príprava prebieha na výpočetných uzloch automaticky. | 3.2.3 |
| R11 | Klient má možnosť pri úlohách špecifikovať požiadavky na výpočetný stroj, ktoré cluster berie do úvahy počas plánovania a rozdeľovania úloh. | 3.2.1, 4.3.2, A.1.1 |
| R12 | Cluster podporuje rozkladanie záťaže podľa predefinovaných pravidiel (*round robin* algoritmus, pravidlá pre uprednostnenie výpočetných strojov, plánovacia priorita). | 3.2.2 |
| R13 | Plánovací algoritmus je konfigurovateľný. | 3.2.2, 4.1 |
| R14 | Cluster je možné jednoducho rozšíriť o ďalšie technológie, pomocou ktorých sa spúšťajú úlohy. | 4.2.1 |

Tabuľka 3.1: Požiadavky na systém.

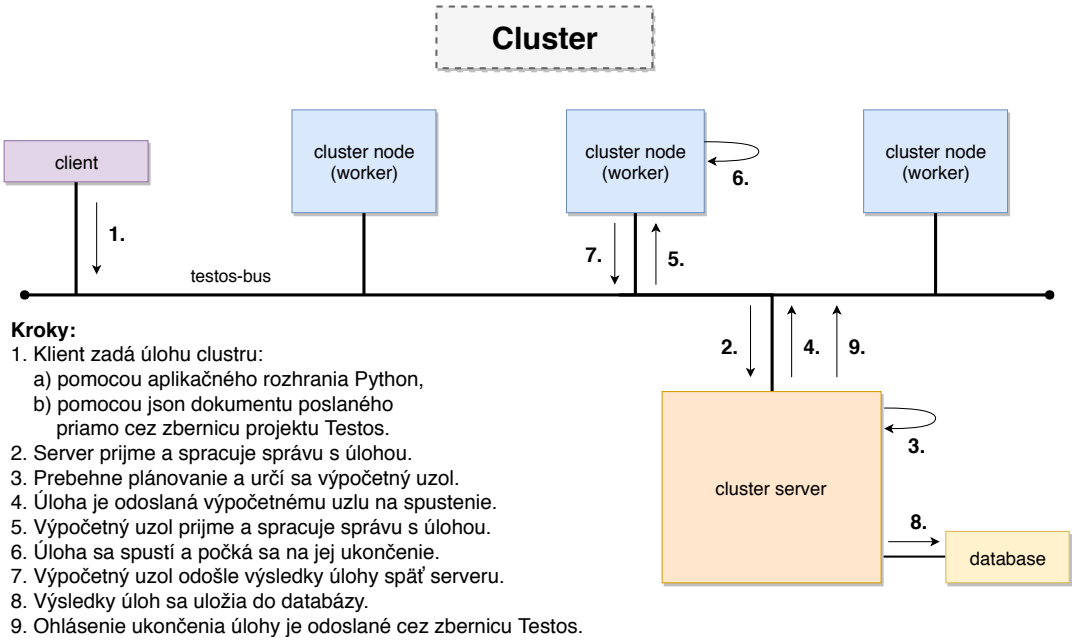
## 3.2 Návrh výpočetného clustru

Cluster, ktorý vznikol ako výsledok tejto práce, dokáže spúšťať úlohy — užívateľom definované skripty — vo virtuálnom prostredí na výpočetných strojoch. Tvoria ho štyri hlavné časti: server, databáza, klienti a výpočetné uzly. Server má na starosti komunikáciu s klientami, rozdeľovanie úloh výpočetným uzlom (plánovanie), ukladanie výsledkov z behu úloh do databázy a správu výpočetných uzlov. Klientom sa v tejto práci rozumie akékoľvek zariadenie, ktoré clustru úlohy zadáva alebo preberá výsledky úloh. Výpočetným uzlom sa myslí akýkoľvek fyzický alebo virtuálny stroj na ktorom beží aplikácia Worker (program vytvorený v rámci tejto práce). Worker má na starosti prípravu prostredia pre beh úlohy, spúšťanie úlohy a nahlasovanie výsledkov späť na server. Každý výpočetný stroj musí podporovať minimálne jednu z technológii virtualizácie, v tejto práci označovanej ako *exekučná platforma*, a každá úloha musí špecifikovať, pre akú exekučnú platformu je určená. Aktuálne je podporovaná iba jedna exekučná platforma, ktorou je Docker, ale návrh systému je prispôsobený pre pridávanie ďalších (napríklad libvirt).

V ďalších podkapitolách sú podrobnejšie popísané tieto časti:

- Úloha, jej definícia a stavy – podkapitola 3.2.1.
- Plánovací algoritmus a rezervácia zdrojov výpočetných uzlov – podkapitola 3.2.2.
- Výpočetné stroje, exekučné platformy a aplikácia Worker – podkapitola 3.2.3.
- Databáza – podkapitola 3.2.4.
- Komunikácia v clustri – podkapitola 3.2.5.

Pre jednoduchšiu predstavu ako sú jednotlivé časti systému prepojené a k akým udalostiam v clustri dochádza po obdržaní úlohy od klienta, je systém spolu so sledom udalostí znázornený na obrázku 3.1.




Obr. 3.1: Sekvencia udalostí v clustri zobrazujúca priechod úlohy systémom.

### 3.2.1 Úlohy

Úloha v tejto implementácii clustru predstavuje dátovú štruktúru pozostávajúcu z nasledujúcich položiek:

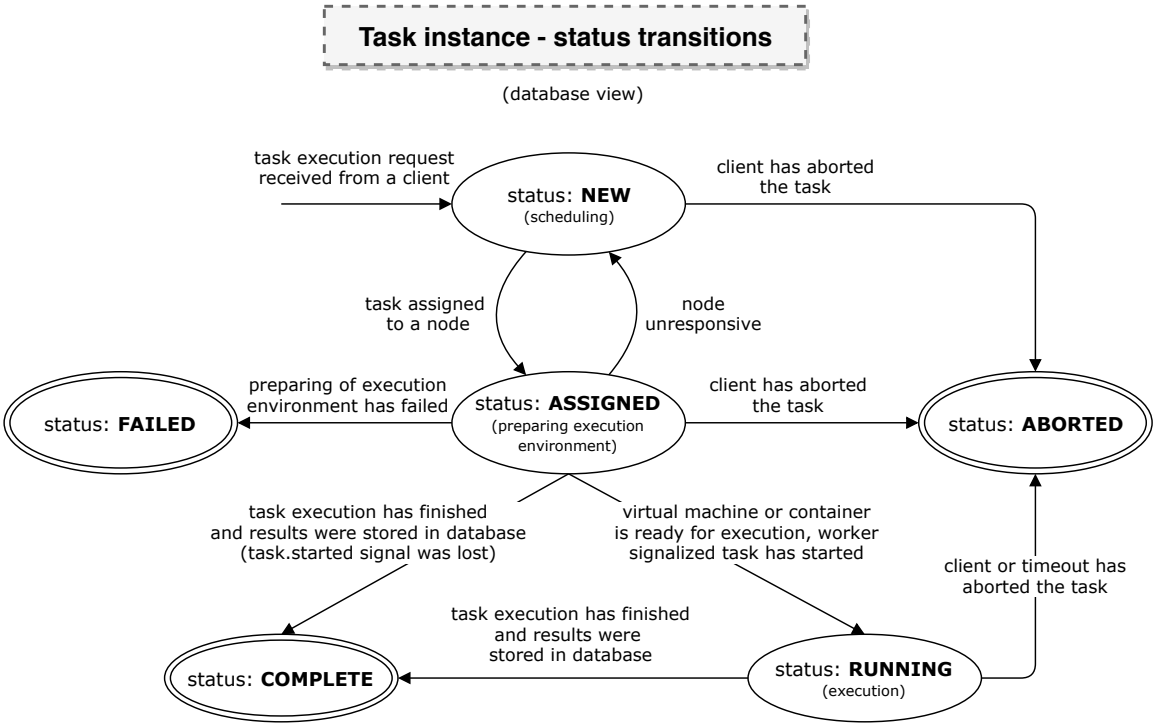
1. skript určený na spustenie,
2. názov exekučnej platformy, pre ktorú je úloha určená,
3. názov obrazu disku, na ktorom má úloha bežať – je špecifický pre konkrétnu exekučnú platformu a spravidla definuje operačný systém, popr. jeho verziu,
4. (voliteľne) vstupné artefakty – súbory, ktoré sa pred spustením úlohy nakopírujú do pracovného adresára,
5. (voliteľne) požiadavky na výpočetný stroj,
6. (voliteľne) pravidlá pre uprednostnenie určitých výpočetných strojov,
7. (voliteľne) parametre skriptu – vysvetlené nižšie,
8. ďalšie parametre úlohy ako je plánovacia priorita, odhadovaný a maximálny čas behu, štítky slúžiace pre kategorizáciu a označovanie úloh.

Príklad špecifikácie úlohy pomocou formátu json je možné vidieť na obrázku A.1 (popis k tomuto obrázku sa nachádza v prílohe A.1.1).

**Parametrizácia úloh** Ak je úloha parametrizovaná, skript sa spustí toľko krát, koľko je parametrov, pričom každé spustenie skriptu dostane ako prvý argument jeden z nich. Jednotlivé spustenia skriptu úlohy spolu s informáciami o behu a s výsledkami sa nazývajú *instanciami úlohy*. Pre všetky instancie sa v databázi vytvára položka hneď po obdržaní úlohy serverom a spustenie jednotlivých instancií nemusí nutne prebiehať na rovnakom stroji. Ak je dostatok výpočetných strojov, ktoré spĺňajú požiadavky definované v úlohe, budú instancie úlohy spúšťané paralelne.

**Instancia úlohy** Každá instancia úlohy sa z pohľadu servera nachádza v určitom stave. Na diagrame nižšie (obr. 3.2) sú zobrazené všetky tieto stavy spolu s prechodmi medzi nimi. Stav `NEW` je počiatočný stav, v ktorom sa instancia nachádza hneď po vytvorení a do stavu `ASSIGNED` je prepnutá po jej priradení plánovacím algoritmom výpočetnému uzlu. Ak výpočetný uzol nepotvrdí prijatie správy do určitého času, instancia sa znovu prepne do stavu `NEW` a uzol sa označí ako neaktívny. Do stavu `ABORTED` sa môže dostať z akéhokoľvek nekoncového stavu a to na podnet klienta alebo po vypršaní timeoutu.

Prípad, kedy sa instancia prepne z `ASSIGNED` priamo do stavu `COMPLETE` by nemal nastávať ale implementačne je možný v prípade, že signál, ktorý mal serveru oznámiť začiatok exekúcie, bol stratený (vysvetlenie signálov v kontexte testos-bus sa nachádza v podkapitole 3.2.5).



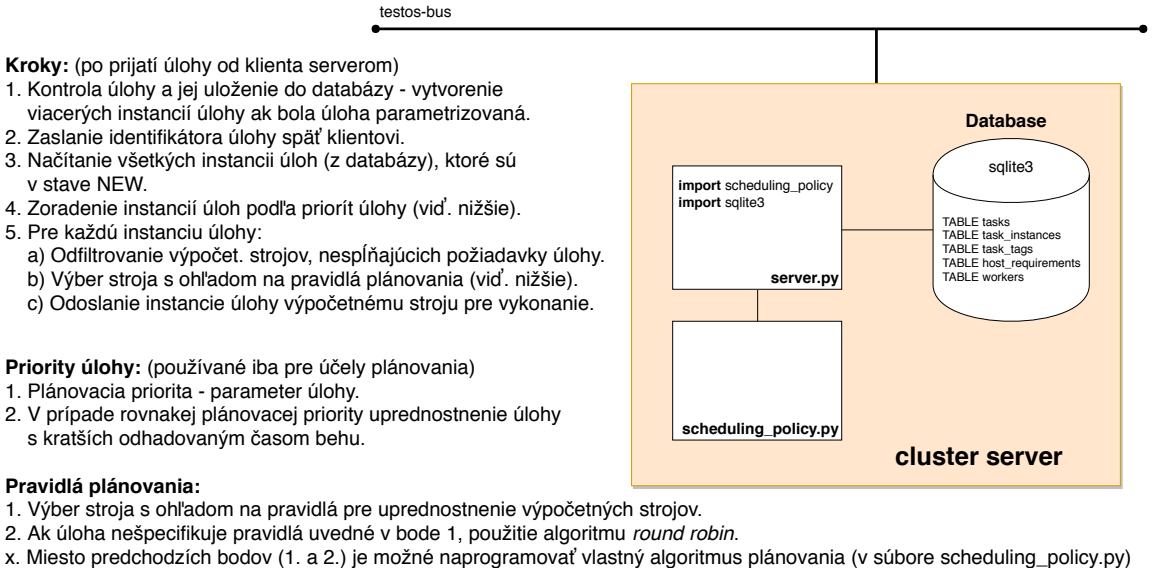

Obr. 3.2: Stavy úlohy z pohľadu databázy.

### 3.2.2 Plánovanie a rezervácia zdrojov

**Plánovanie** Keďže cluster má k dispozícii spravidla viac ako jeden výpočetný uzol, musí obsahvať logiku pre rozhodovanie:

- ktorá instancia úlohy sa bude spúšťať na ktorom výpočetnom stroji,
- ktoré úlohy budú spustené prioritne.

Na obrázku 3.3 je zobrazená sekvencia krokov zobrazujúca plánovací algoritmus, spustený po tom, ako server obdržal novú úlohu od klienta. V aktuálnej implementácii *pravidiel plánovania*, ktoré sú na obrázku rozpísané, sa spomínajú *pravidlá pre uprednostnenie výpočetných strojov*. Jedná sa o pole atribútov výpočetných strojov (atribúty podrobnejšie vysvetlené v podkapitole 3.2.3), podľa ktorých sa stroje pred výberom zoradia (zostupne), pričom sa dbá na poradie, v ktorom sú atribúty v poli uvedené (atribúty na menšom indexe v poli majú väčšiu prioritu pri zoraďovaní). Po tomto zoradení sa vezme prvý výpočetný stroj zo zoznamu a použije sa pre výpočet danej úlohy. Časť plánovacieho algoritmu, ktorá je na obrázku označená ako `5b`, obsahuje logiku pre výber výpočetného stroja po tom, ako boli odfiltrované stroje nespĺňajúce požiadavky úlohy. Táto časť je konfigurovateľná – to znamená, že si administrátor clustru môže napísať vlastný algoritmus, ktorý ju upraví/nahradí (návod v podkapitole 4.1).




Obr. 3.3: Zjednodušený algoritmus plánovača.

Na obrázku 3.3 bol plánovací algoritmus spustený udalosťou príchodu novej úlohy od klienta. Existuje niekoľko ďalších udalostí, ktoré vyvolajú spustenie plánovania:

- pripojenie nového výpočetného uzlu,
- vypršanie časovača pri čakaní na potvrdenie od výpočetného uzlu po tom, ako mu bola zaslaná úloha na exekúciu (daná instancia úlohy sa v tomto prípade prepne znovu do stavu `NEW`)
- výpočetný uzol nahlásil ukončenie behu úlohy.

**Rezervácia zdrojov** Každá úloha si môže definovať zdroje, ktoré potrebuje pre svoj beh. Cluster rozlišuje tri zdroje, ktoré je možné rezervovať: jadrá procesoru, operačnú pamäť a grafický procesor. Rezervácia týchto zdrojov však v aktuálnej implementácii clustru prebieha iba na strane servera, tzn. úlohy bežiace na výpočetných uzloch reálne nemajú rezerované žiadne zdroje (vysvetlenie v nasledujúcom odseku). Na serveri je rezervácia zdrojov z dôvodu obmedzenia počtu a typov úloh, ktoré na výpočetných uzloch môžu bežať súčasne a zároveň dáva možnosť výpočetným uzlom tento počet do istej miery ovplyvňovať.

Server si teda udržuje informáciu o voľných zdrojoch každého výpočetného uzlu a úlohy prideľuje iba tým uzlom, ktorí majú dostatok voľných zdrojov, aby uspokojili požiadavky danej úlohy. Každá úloha vyžaduje minimálne 1 jadro procesoru a to aj v prípade, že tento požiadavok klient explicitne nedefinoval pri jej zadávaní. Zároveň nie je možné pri zadávaní úlohy definovať nulové hodnoty požiadavkov na jadrá procesoru a operačnú pamäť. Ak sa o to klient pokúsi, server úlohu odmietne.

**Rezervácia zdrojov na výpočetných uzloch** Na výpočetných strojoch sa zdroje nerezervujú kvôli menšiemu zaťaženiu týchto počítačov a teda aplikácia Worker môže bežať aj na počítačoch, ktorých primárny účel nie je byť súčasťou výpočetného clustru. Je to kvôli tomu, že úloha nemusí využívať dané zdroje po celý čas svojho behu a tým pádom by boli rezervované zbytočne. Keďže sa neočakáva, že sa na výpočetných uzloch budú spúšťať

služby a aplikácie vyžadujúce vysokú dostupnosť, je prijateľné, že sa zdroje rezervujú až keď o ne bude žiadané (v prípade procesorového času – keď plánovač pridelí procesu jadro procesora). Úlohy ale takisto nie sú nijak obmedzované v žiadaní o zdroje počas ich behu, takže je na klientoch, aby definovali správne hodnoty požiadavkov na výpočetné stroje pri vytváraní úloh.

### 3.2.3 Výpočetné stroje a spúšťanie úloh

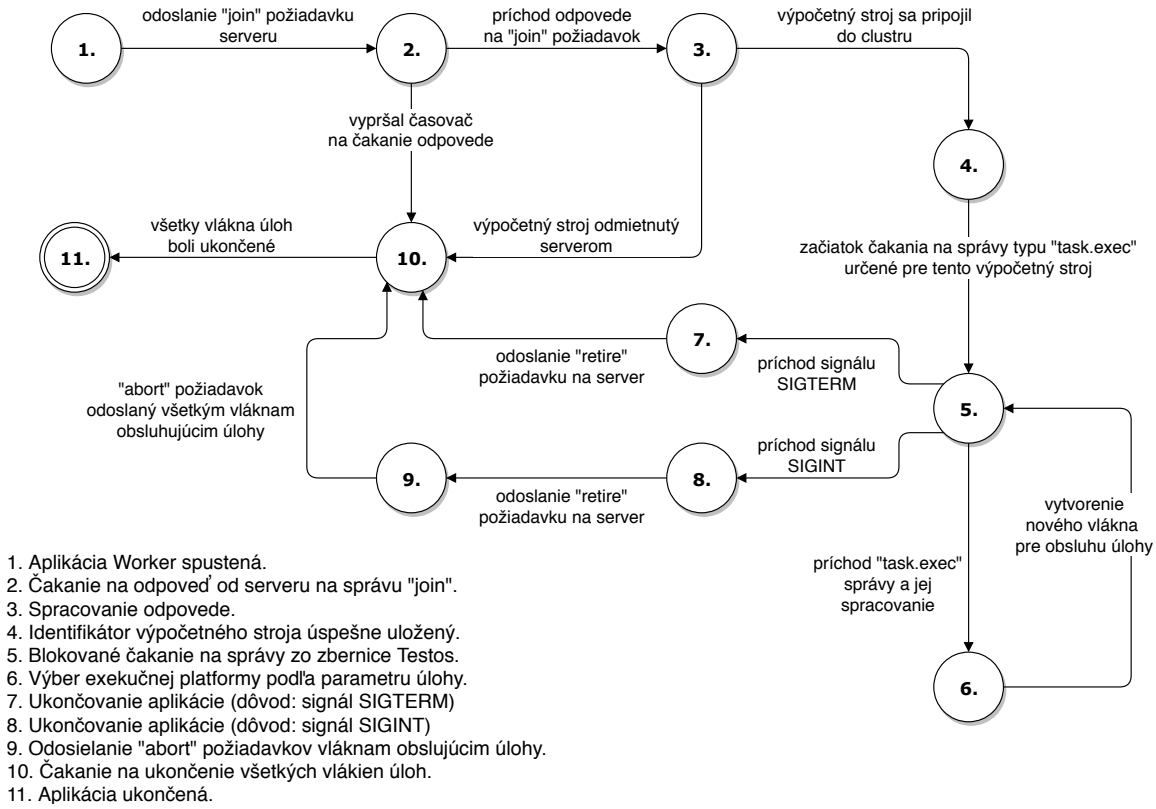
**Exekučné platformy** Aplikácia Worker aktuálne podporuje iba jednu exekučnú platformu – docker. Cluster je ale navrhnutý tak, že je možné pridávať ďalšie, pričom s možnosťou nasledujúcich dvoch sa počítalo už pri návrhu:

- *libvirt*[1] – open-source knižnica pre ovládanie virtualizačných platforiem ako je KVM alebo QEMU.
- *nvidia-docker* – jedná sa o nadstavbu na docker príkaz, ktorá sa používa pre zjednodušenie procesu sprístupnenia grafických procesorov (znakových zariadení, užívateľských knižníc a ovládača) v docker kontajneri. Projekt spadá priamo pod výrobcu Nvidia a vývojári nvidia-docker k uľahčeniu tohto procesu poskytujú aj špeciálne obrazy disku CUDA [15].

**Pripájanie výpočetných uzlov k serveru** Každá aplikácia Worker pri pripájaní k serveru nahlasuje parametre výpočetného stroja. Zoznam všetkých parametrov, ktoré server od výpočetného uzlu očakáva, sa nachádza v tabuľke 4.1 (parametre zo sekcií `testos-bus` a `general` sú výnimkou a slúžia iba na nastavenie chodu aplikácie).

**Životný cyklus aplikácie Worker** Na obrázku 3.4 je zobrazený životný cyklus hlavného vlákna aplikácie. Každá aplikácia Worker sa pripája práve k jednému serveru, pričom na výpočetnom uzli môže takýchto aplikácii bežať viacero (za predpokladu že používajú odlišný kanál pre komunikáciu na zbernici Testos). Prípad, kedy by užívateľ mohol chcieť viacero bežiacich aplikácií ovládajúcich výpočetný stroj, by mohol nastať, ak by chcel stroj pripojiť súčasne k dvom rôznym clustrom. Pokiaľ existuje iba jeden cluster server, nie je potreba aplikáciu spúšťať viac krát – aplikácia dokáže spúšťať úlohy a ovládať ich beh paralelne (viď vytváranie vlákien medzi stavmi 5 a 6 na obrázku).

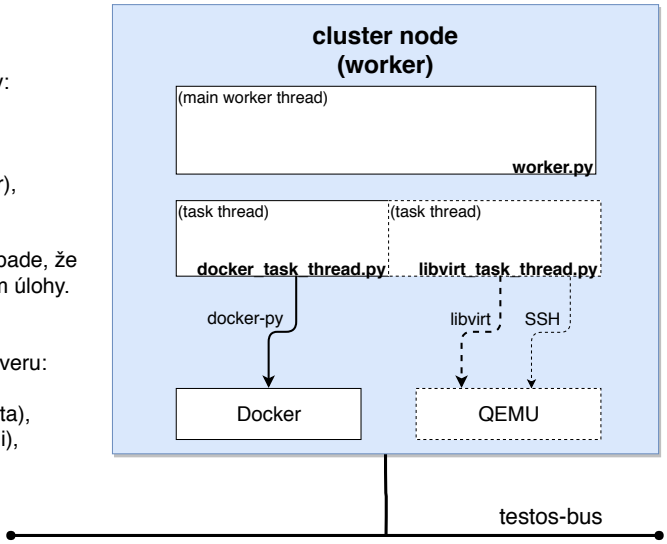
[1] https://libvirt.org/




Obr. 3.4: Životný cyklus aplikácie Worker.

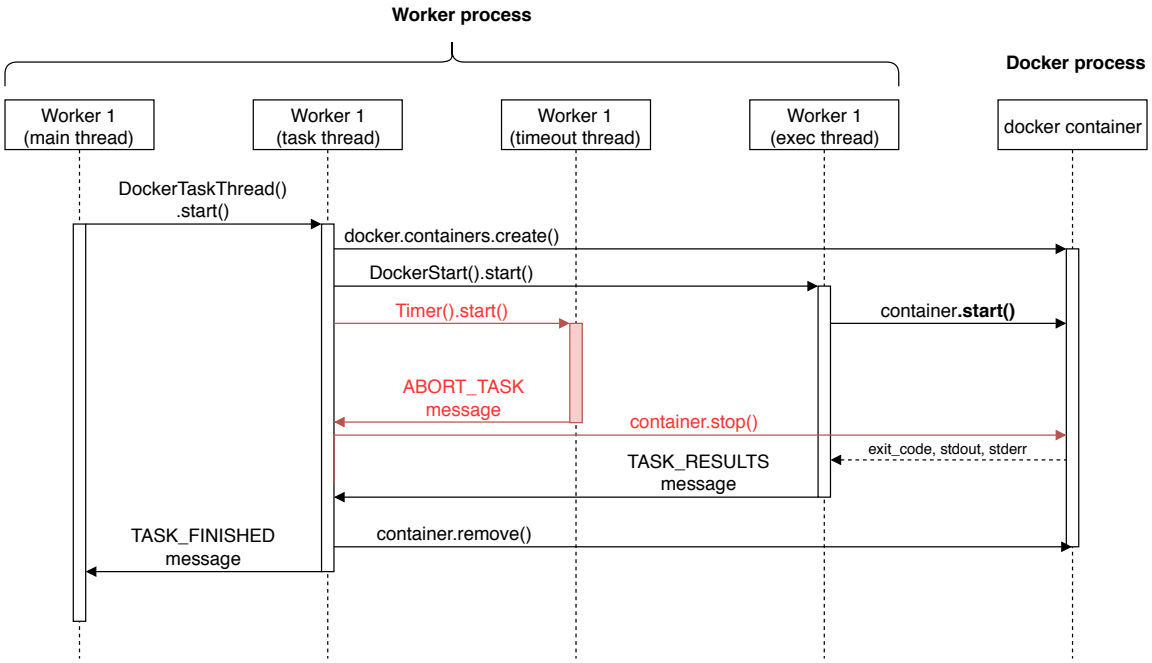
**Spúšťanie úloh a vlákna aplikácie** Na obrázku 3.5 je popísaný algoritmus spustenia úlohy po jej prevzatí výpočetným uzlom. Časť aplikácie, ktorej okraje sú nakreslené prerušovanou čiarou je časť, ktorá zatiaľ nie je implementovaná.
Keďže je aplikácia Worker navrhnutá multivláknovo, dokáže úlohy spúšťať paralelne. Na obrázku 3.6 môžeme vidieť vytváranie/zanikanie vlákien a procesov pri spúšťaní úlohy v prípade použitia exekučnej platformy docker. Časť nakreslená červenou farbou sa vykonáva iba v prípade, že úloha má určenú maximálnu dobu svojho behu (a teda je potrebné spustiť časovač, ktorý úlohu po tejto dobe preruší). Na diagrame je zobrazený prípad, kedy časovač vyvolal zastavenie kontajneru pred skončením úlohy v ňom bežiacej (a tým jej zrušenie).

**Kroky:**
1. Kontrola úlohy a načítanie jej parametrov.
2. Vytvorenie nového vlákna pre obsluhu úlohy.
3. (vlákno úlohy) Príprava prostredia pre beh úlohy:
   - docker: vytvorenie a nastavenie kontajneru,
   - libvirt: vytvorenie virtuálneho stroja.
4. (vlákno úlohy) Pripojenie na vytvorený stroj:
   - pomocou knižnice docker-py (v prípade docker),
   - pomocou SSH (v prípade libvirt).
5. *Spustenie úlohy.
6. *Spustenie časovača pre prerušenie úlohy v prípade, že beh prekročí časovú hranicu určenú parametrom úlohy.
7. (vlákno úlohy) Zhromaždenie výsledkov úlohy a ich komprimácia.
8. (vlákno úlohy) Nahlasovanie výsledkov späť serveru:
   a) výsledky behu úlohy (štandardný výstup, štandardný chybový výstup, návratová hodnota),
   b) výstupné artefakty (súbory určené k archivácii),
   c) **(voliteľne) vrátenie celého obrazu disku,
   d) štatistiky z behu úlohy:
      - doba behu (reálny čas),
      - **maximálne využitie pamäte,
      - **procesorový čas.




* Činnosť nesmie zablokovať vlákno úlohy vytvorené v bode 2.
** Zatiaľ neimplementované.

Obr. 3.5: Algoritmus spustenia úlohy na výpočetnom uzle.




Obr. 3.6: Komunikácia medzi vláknami pri exekúcii úlohy (docker).

**Odpájanie výpočetných uzlov z clustru** Pre odpojenie výpočetného uzlu z clustru je potrebné ukončiť aplikáciu Worker. Na obrázku 3.4 zobrazujúcom životný cyklus aplikácie môžme vidieť, že toto sa dá dosiahnúť dvoma spôsobmi a to zaslaním signálov[2]:

- `SIGTERM` – v tomto prípade aplikácia oznámi serveru, že už neprijíma žiadne ďalšie úlohy (zaslaním *retire* požiadavku serveru - viď. príloha A.4.2) a počká na ukončenie všetkých rozpracovaných úloh.
- `SIGINT` – podobne ako v predošlom prípade, aplikácia oznámi serveru, že už neprijíma nové úlohy, pričom aktuálne bežiace úlohy vynútene ukončí (rovnakým spôsobom, ako by o ich ukončenie požiadal klient).

### 3.2.4 Databáza

V návrhu projektu Testos je zahrnuté vytvorenie vlastného databázového úložiska, ktoré by mal v budúcnosti používať aj cluster vytvorený v rámci tejto práce. Toto úložisko ale v čase implementácie clustru ešte nie je hotové, preto sa v projekte aktuálne používa SQLite[3] databáza. Je to jedna z odľahčených (zjednodušených) verzií transakčných databázových systémov a je spravovaná výlučne kódom aplikácie (teda nejedná sa o databázu, ktorá by bežala v oddelenom procese).

V databázi sú uložené úlohy, ich výsledky a informácie o výpočetných uzloch. Zdrojový kód clustru obsahuje všetky potrebné SQL príkazy k jej vytvoreniu a správe, takže pri prvom spustení serverovej aplikácie sa databáza vytvorí automaticky – je k tomu ale potreba nakofigurovať cestu k súboru, v ktorom bude uložená. K tomu slúži kľúč `db_file` v konfiguračnom súbore (príklad konfiguračného súboru sa nachádza v prílohe B.1), ktorý môže obsahovať absolútnu alebo relatívnu cestu. V prípade relatívnej cesty sa uvádza cesta relatívna ku koreňovému priečinku projektu.

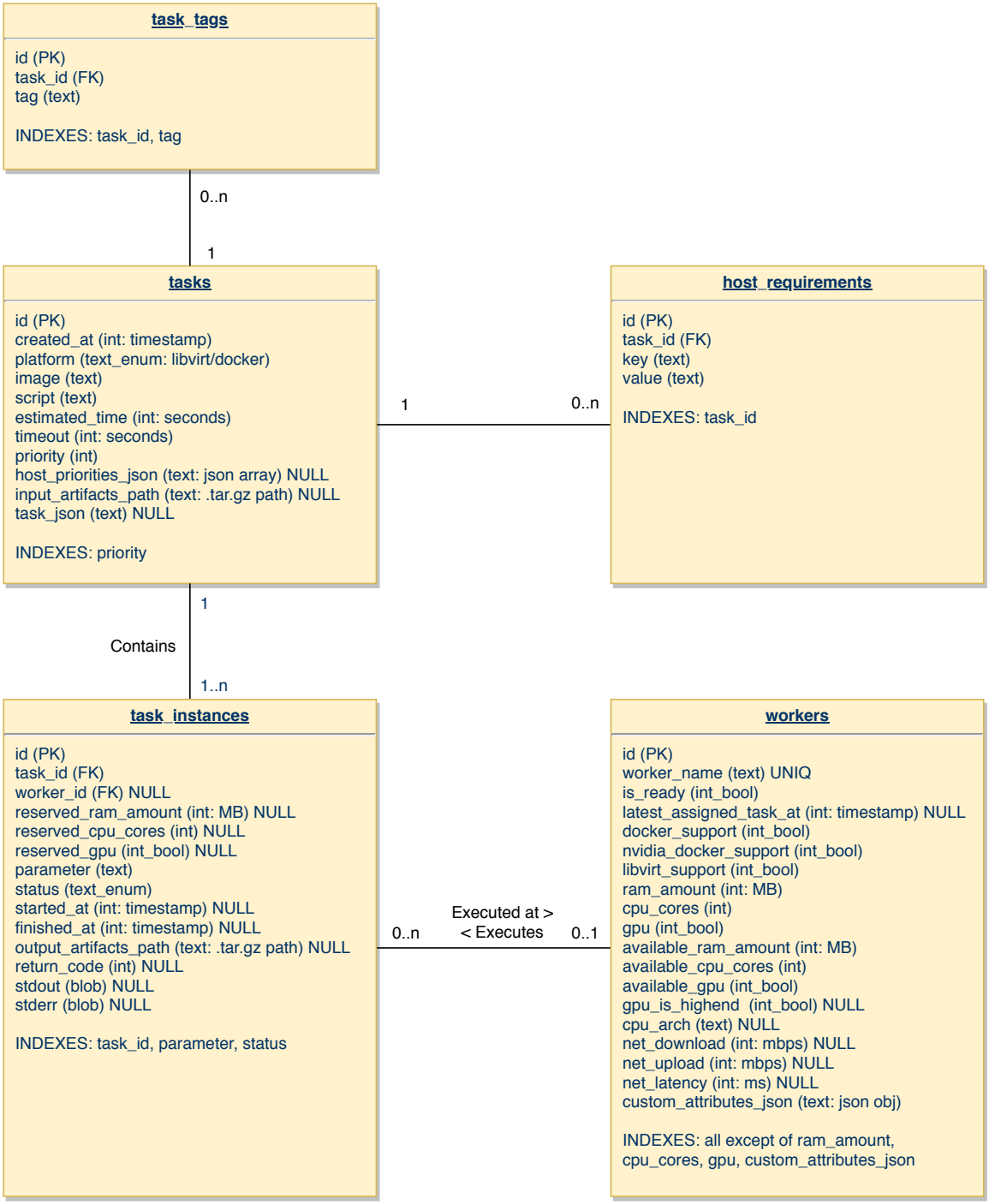
Vstupné a výstupné artefakty sa neukladajú v databáze priamo ale ako súbory na disku a do databázy sa následne ukladá cesta k týmto súborom. Z toho dôvodu je potrebné v konfiguračnom súbore nastaviť hodnotu `artifacts_dir`, ktorá by mala obsahovať cestu k priečinku v súborovom systéme serveru, do ktorého sa artefakty úloh budú ukladať.

Na obrázku 3.7 môžme vidieť diagram zobrazujúci tabuľky databázy a ich prepojenie. V každom obdĺžniku znázorňujúcom tabuľku sú vymenované stĺpce, pričom v zátvorke je za názvom stĺpcu vždy uvedený dátový typ. Ich význam je nasledovný:

- `text` – textový reťazec (základný dátový typ SQLite: TEXT). V zátvorke môže za typom nasledovať dvojbodka a význam hodnoty daného stĺpca.
- `text_enum` – textový reťazec s použitím obmedzenia hodnoty iba na konkrétne reťazce (vymenované za dvojbodkou).
- `int` – číslo (základný dátový typ SQLite: INTEGER). V zátvorke môže za typom nasledovať dvojbodka a jednotky, v ktorých sa ukladajú hodnoty do daného stĺpca.
- `int_bool` – číselný dátový typ; stĺpec by mal obsahovať iba hodnoty 0 a 1.
- `blob` – binárny reťazec/objekt.

[2] signálom sa tu myslí medziprocesová komunikácia na systémoch UNIX
[3] https://docs.python.org/3/library/sqlite3.html

- `PK` – skratka pre primárny kľúč tabuľky - vo všetkých prípadoch sa jedná o celé číslo (INTEGER).
- `FK` – skratka pre cudzí kľúč.




Obr. 3.7: Cluster databáza (sqlite).

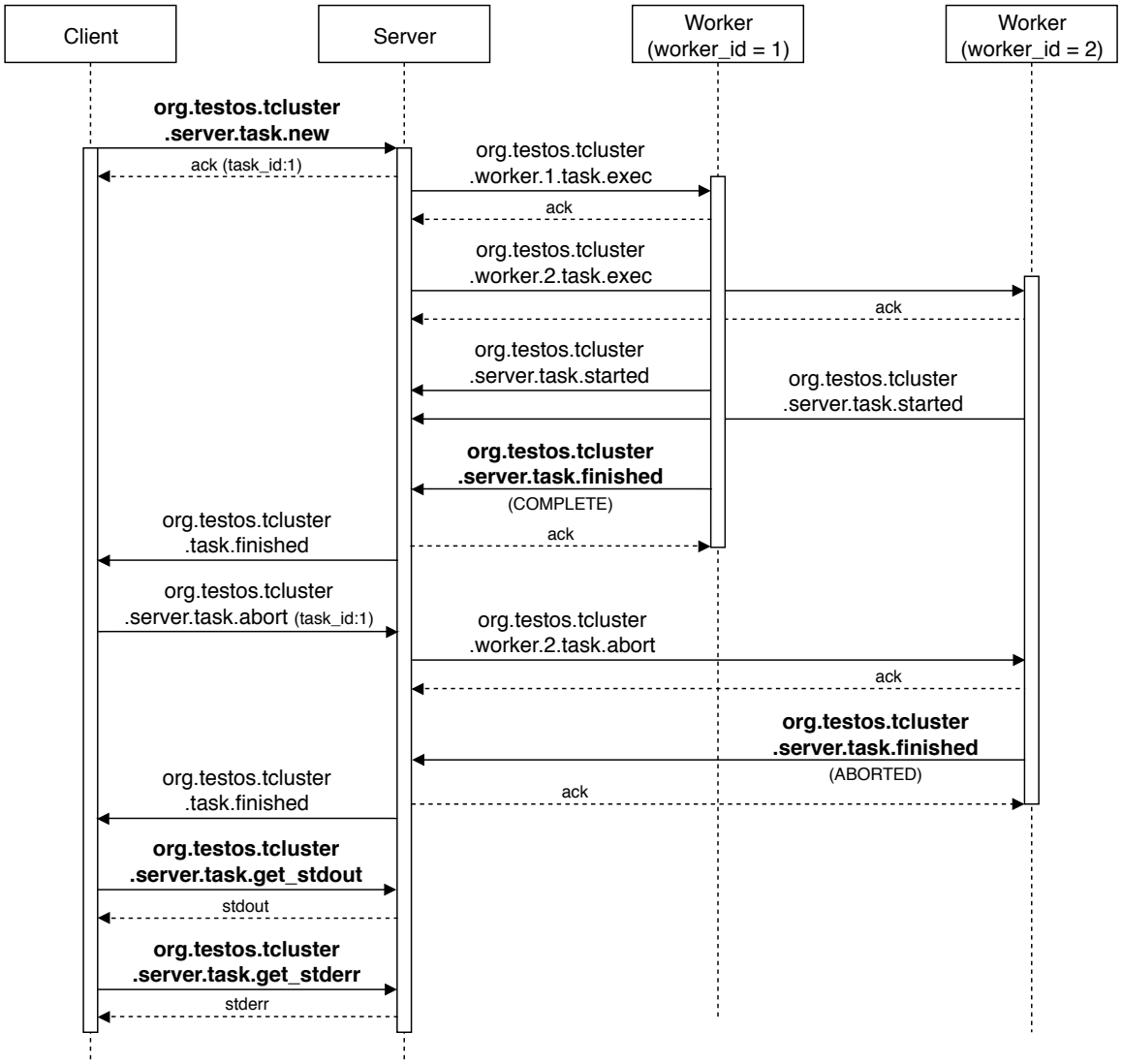
### 3.2.5 Komunikácia cez zbernicu Testos

Všetka komunikácia medzi zariadeniami v clustri prebieha cez zbernicu Testos. V clustri nedochádza ku komunikácii medzi klientami a výpočetnými uzlami, takže vo všetkých správach je server buď v úlohe odosielateľa alebo príjemcu.

**Druhy správ** Zbernica projektu Testos umožňuje posielať dva druhy správ, pričom oba sú v clustri využívané:

- *Request* – jedná sa o správu, na ktorú odosielateľ očakáva odpoveď.
- *Signal* – správa bez odpovede.

Na obrázku 3.8 je znázornený sled správ posielaných po zbernici Testos tak, ako by v clustri prebiehali po tom, čo server obdrží novú úlohu od klienta. Klient v tomto prípade úlohu parametrizoval, preto na obrázku môžme vidieť odosielanie dvoch instancií úloh výpočetným uzlom. Správy vyznačené hrubým písmom sú odosielané v blokujúcom móde - to znamená, že k návratu riadenia do kódu dôjde až po obdržaní odpovede od klienta. Zoznam všetkých správ posielaných v rámci clustru sa nachádza v prílohe A.




Obr. 3.8: Testos-bus správy prebiehajúce v clustri pro obdržaní úlohy od klienta.

# Kapitola 4

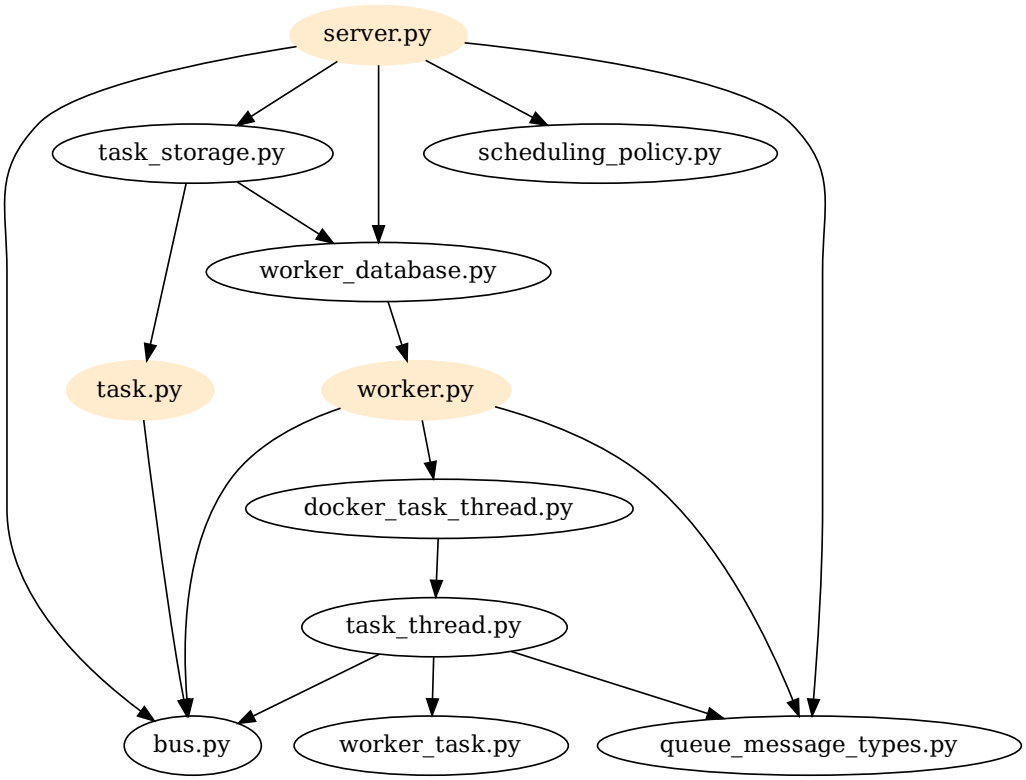
# Implementácia a testovanie clustru

Pre implementáciu všetkých častí výsledného systému bol zvolený jazyk *Python 3*. Na obrázku 4.1 je znázornený graf závislostí medzi zdrojovými súbormi projektu.

Stručný popis jednotlivých súborov:

- `server.py` - serverová aplikácia.
- `task_storage.py` - modul pre prácu s úložiskom úloh a ich výsledkov na strane serveru (aktuálne sa jedná o rozhranie pre prácu s SQLite databázou).
- `scheduling_policy.py` - súbor obsahujúci konfigurovateľnú časť algoritmu plánovania.
- `worker_database.py` - modul pre prácu s úložiskom informácií o výpočetných uzloch na strane serveru (aktuálne sa jedná o rozhranie pre prácu s SQLite databázou).
- `worker.py` - aplikácia pre ovládanie výpočetných uzlov.
- `task.py` - súbor implementujúci triedu *Task*, ktorá slúži ako klientské rozhranie clustru v jazyku Python a zároveň je používaná aj modulom pre správu úložiska úloh.
- `docker_task_thread.py` - modul implementujúci triedu *DockerTaskThread*, ktorá sa používa pre vytváranie vlákien obsluhujúcich úlohy určené pre exekučnú platformu `docker`.
- `task_thread.py` - súbor obsahujúci abstraktnú triedu *TaskThread*, ktorú by mali dediť všetky triedy implementujúce obsluhu pre jednotlivé exekučné platformy (viac v podkapitole 4.2.1).
- `bus.py` - modul poskytujúci *Fake*[1] objekty zbernice Testos určené pre testovanie projektu.
- `worker_task.py` - súbor implementujúci triedu *WorkerTask*, používanú všetkými triedami typu *TaskThread*.
- `queue_message_types.py` - modul obsahujúci zoznam typov správ posielaných medzi vláknami v implementovaných aplikáciach.

[1] https://en.wikipedia.org/wiki/Mock_object#Mocks,_fakes,_and_stubs




Obr. 4.1: Graf závislostí medzi zdrojovými súbormi projektu.

## 4.1 Serverová aplikácia

Serverová aplikácia sa dá spustiť pomocou nasledujúceho príkazu v koreňovom priečinku projektu:

```
make run-server
```

alebo pomocou priameho spúšťania súboru `server.py` (v tomto prípade sa dá definovať cesta ku konfiguračnému súboru):

```
src/server.py --config=server.ini
```

V oboch prípadoch spustenia sa dá ukončiť zaslaním signálu `SIGTERM` alebo `SIGINT`.

Štruktúra priečinku, v ktorej si serverová aplikácia ukladá dáta počas svojho behu vyzerá nasledovne (čísla v priečinku `task_artifacts` značia identifikátory instancií úloh z databázy):

```
runtime/
└── server
    ├── database
    │   └── tcluster.db
    └── task_artifacts
        ├── 1
        │   └── input_artifacts.tgz
        └── 2
            ├── input_artifacts.tgz
            └── output_artifacts.tgz
```

**Konfigurácia plánovacieho algoritmu** Ak by chcel správca výpočetného clustru upraviť časť plánovacieho algoritmu, ktorá je na obrázku 3.3 označená ako `5b`, môže tak spraviť pomocou implementácie vlastnej funkcie *decide_worker()* v súbore `scheduling_policy.py`. Funkcia musí byť napísaná v jazyku *Python 3* a musí definovať dva argumenty: prvý je objekt typu *dict* obsahujúci instanciu úlohy, ktorá je aktuálne v procese plánovania a druhý je objekt typu *list* obsahujúci výpočetné uzly, ktoré spĺňajú požiadavky úlohy a aktuálne majú dostatok voľných zdrojov pre jej spustenie. Súbor obsahuje v komentároch príklady argumentov, ktoré budú funkcii predané v čase jej volania. Očakávaná návratová hodnota funkcie je identifikátor výpočetného uzlu, ktorý bude zvolený pre počítanie úlohy.

## 4.2 Aplikácia pre výpočetné stroje

Aplikácia sa dá spustiť pomocou nasledujúceho príkazu zadaného v koreňovom priečinku projektu:

```
make run-worker
```

alebo pomocou priameho spúšťania súboru `worker.py` (v tomto prípade sa dá definovať cesta ku konfiguračnému súboru):

```
src/worker.py --config=worker.ini
```

Štruktúra priečinku, v ktorej si aplikácia Worker ukladá dáta počas svojho behu vyzerá nasledovne (čísla podadresárov v adresári `worker` značia identifikátory instancií úloh):

```
runtime/
└── worker
    └── 1
        ├── output_artifacts
        │   └── task.artifact
        └── workdir
            ├── combinations.example.txt
            ├── entry_point.sh
            └── output_artifacts
```

Zoznam všetkých parametrov nastaviteľných v konfiguračnom súbore aplikácie sa nachádza v tabuľke 4.1 (je nutné ich nastaviť ešte pred jej spustením). Ukážka konfiguračného súboru sa nachádza v prílohe B.2. Vysvetlivky k niektorým z parametrov uvedených v tabuľke:

- `default_cluster_channel` – základný komunikačný kanál, pre komunikáciu cez zbernicu Testos. Musí byť nastavený na rovnakú hodnotu, akú používa server.
- `runtime_dir` – umiestnenie v súborovom systéme výpočetného stroja, ktoré aplikácia použije pre vytváranie dočasných pracovných adresárov pre úlohy. Môže predstavovať absolútnu alebo relatívnu cestu. V prípade relatívnej cesty sa uvádza cesta relatívna ku koreňovému priečinku projektu.
- `log_level` – úroveň záznamu. Jedna z hodnôt: `ERROR`, `WARNING`, `INFO`, `DEBUG`.
- `ram_amount` – dostupná operačná pamäť zadávaná v jednotkách MB. Užívateľ nemusí zadať reálnu veľkost operačnej pamäte na stroji ale mal by mať na pamäti, že daná hodnota sa použije pre obmedzenie počtu úloh, ktoré na výpočetnom stroji môžu bežať (viď. odsek *Rezervácia zdrojov* v podkapitole 3.2.2).
- `cpu_cores` – počet jadier procesora. Hodnota sa — takisto ako veľkosť operačnej pamäte — použije pre obmedzenie počtu paralelne bežiacich úloh.
- `gpu` – parameter signalizujúci prítomnosť grafického procesora. Rovnako ako predchodzie dva parametre, sa tento parameter používa pre obmedzenie počtu súčasne bežiacich úloh vyžadujúcich grafický procesor. Keďže tento parameter je typu boolean, povoľuje beh buď jednej alebo žiadnej takejto úlohy.
- `gpu_is_highend` – oznamuje, či je na výpočetnom stroji vysokovýkonný grafický procesor. Opakom je integrovaný grafický čip.
- `worker_name` – názov výpočetného stroja, ktorý musí byť v rámci clustru jedinečný. Ak užívateľ použije názov, ktorý sa už v clustri nachádza a stroj, ktorý ho používa, je práve aktívny, server príchodzí požiadavok výpočetného stroja na pripojenie do clustru zamietne. Ak názov v clustri existuje ale stroj, ktorý ho používa aktuálne nie je aktívny, server požiadavok prijme a aktualizuje parametre stroja v databáze.
- `*` (asterisk) v tabuľke znamená akýkoľvek reťazec. V sekcii `custom-attributes` si užívateľ môže definovať vlastné parametre výpočetného stroja, ktoré potom môžu byť vyžadované úlohami. Tieto parametre môžu byť iba typu *boolean*.

| parameter výpočetného stroja | Odpovedajúca položka v konfiguračnom súbore: sekcia | kľúč | dátový typ |
|---|---|---|---|
| Základný komunikačný kanál. | testos-bus | default_cluster_channel | string |
| Priečinok pre ukladanie runtime súborov. | general | runtime_dir | string |
| Úroveň záznamu. | general | log_level | string |
| Veľkosť pamäte RAM. | reservable-resources | ram_amount | integer [MB] |
| Počet jadier procesora. | reservable-resources | cpu_cores | integer |
| Prítomnosť GPU. | reservable-resources | gpu | boolean |
| Podpora exekučnej platformy docker. | attributes | docker_support | boolean |
| Podpora exekučnej platformy nvidia-docker. | attributes | nvidia_docker_support | boolean |
| Podpora exekučnej platformy libvirt. | attributes | libvirt_support | boolean |
| Vysokovýkonné GPU. | attributes | gpu_is_highend | boolean |
| Architektúra procesoru. | attributes | cpu_arch | string |
| Rýchlosť siete - sťahovanie. | attributes | net_download | integer [mbps] |
| Rýchlosť siete - nahrávanie. | attributes | net_upload | integer [mbps] |
| Odozva siete. | attributes | net_latency | integer [ms] |
| Názov výpočetného uzlu v rámci clustru. | attributes | worker_name | string |
| Voliteľné atribúty. | custom-attributes | * | boolean |

Tabuľka 4.1: Konfigurácia výpočetného uzlu.

Po príchode úlohy výpočetnému uzlu sa kontroluje exekučná platforma, pre ktorú je úloha určená. Na základe tejto informácie sa vytvorí objekt odpovedajúcej triedy (vetvenie sa nachádza v metóde *run()* v triede *Worker*), ktorá definuje beh vlákna pre obsluhu danej úlohy. Toto vlákno sa následne spustí a hlavné vlákno aplikácie Worker pokračuje v čakaní a spracovávaní správ zo zbernice Testos.

### 4.2.1 Trieda TaskThread

Trieda *TaskThread* bola vytvorená za účelom modularizovať aplikáciu Worker takým spôsobom, aby bolo jednoduché časom pridávať podporu pre nové exekučné platformy. Táto trieda obsahuje niekoľko abstraktných metód, ktoré by mali byť implementované vo viac špecifických triedach. Rozhranie týchto metód je bližšie popísané v dokumentačných reťazcoch jazyka *Python* priamo v zdrojových kódoch.

Zoznam abstraktných metód:

- `prepare_execution_environment()`
- `run_task()`
- `abort_task()`
- `clean()`

### 4.2.2 Trieda DockerTaskThread

Trieda *DockerTaskThread* je trieda, ktorá implementuje obsluhu a spúšťanie úloh určených pre exekučnú platformu `docker`. Pred jej implementáciou bolo potrebné rozhodnutie, akým spôsobom bude aplikácia Worker komunikovať s platformou Docker. Na výber boli dve knižnice:

- *subprocess*[2] – knižnica používaná na vytváranie nových procesov a komunikáciu s týmito procesmi pomocou rúr (angl. *pipes*).
- *docker-py*[3] – knižnica pre komunikáciu s aplikáciou Docker pomocou objektov jazyka Python a ich metód.

Na blogu [4] sa píše o viacerých problémoch s knižnicou *docker-py* ale keďže článok na tomto blogu bol zverejnený v roku 2014, ukázalo sa, že väčšina týchto problémov sa v aktuálnej verzii knižnice (3.3.0) nenachádza – z toho dôvodu som sa rozhodol túto knižnicu uprednostniť.

## 4.3 Klientské rozhranie

Klientské rozhranie slúži na zadávanie nových úloh clustru, rušenie týchto úloh, popr. preberanie ich výsledkov a výstupných artefaktov po ukončení.

### 4.3.1 Rozhranie zbernice Testos

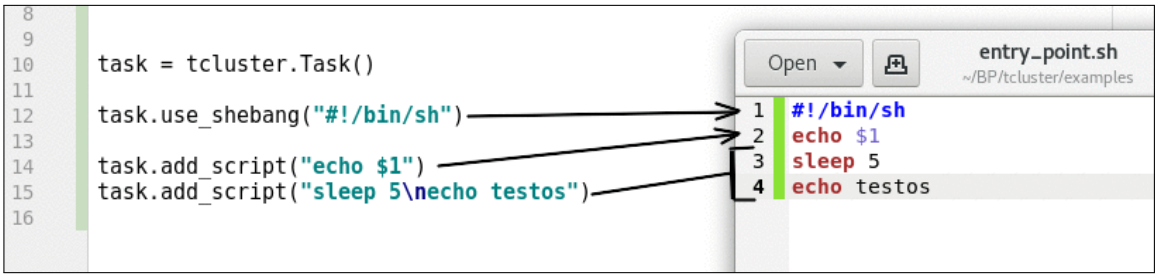
Klient môže serveru zadávať nové úlohy pomocou správ na zbernici Testos. Cez túto zbernicu môže taktiež preberať ich výsledky. Zoznam správ, ktorým server rozumie sa nachádza v prílohe A.

### 4.3.2 Aplikačné rozhranie Python

Aplikačné rozhranie Python sa nachádza v súbore `task.py` a je zdokumentované pomocou dokumentačných reťazcov jazyka Python. Využíva zbernicu projektu Testos, takže slúži ako abstrakcia komunikácie so serverom po tejto zbernici.

Na obrázku 4.2 sa nachádza príklad vytvárania skriptu pomocou metód objektu *Task*, ktorý bude po odoslaní serveru spustený na výpočetnom uzle.

[2] https://docs.python.org/3/library/subprocess.html
[3] https://docker-py.readthedocs.io/




Obr. 4.2: Vytváranie súboru `entry_point.sh`.

## 4.4 Testovacia sada

Pre vytvorenie testovacej sady bol použitý "framework" behave[4].
Testy sa spúšťajú pomocou nasledujúceho príkazu v koreňovom priečinku projektu:

```
$ make test
```

V prípade absencie Docker je možnosť vynechať testy, ktoré vyžadujú aby bol Docker nainštalovaný a spustený:

```
$ make test-without-docker
```

[4] https://behave.readthedocs.io/

# Kapitola 5

# Závěr

Cieľom tejto práce bolo analyzovať požiadavky na systém pre spúšťanie paralelných úloh v projekte Testos, navrhnúť tento systém a následne ho implementovať a zdokumentovať. Toto sa podarilo aj napriek tomu, že výsledný systém je závislý od zbernice projektu, ktorá zatiaľ implementovaná nie je. Z toho dôvodu je možné výsledný systém demonštrovať iba po častiach a nie ako celok.
Počas návrhu a implementácie systému vzniklo viacero námetov na jeho rozšírenie, medzi ktoré patrí napríklad:

- možnosť návratu celého obrazu disku po skončení úlohy klientovi,
- pravidelné kontaktovanie servera výpočetnými uzlami s aktualizáciou informácií o stave siete,
- auto-detekcia hardwaru pri spúšťaní výpočetných uzlov,
- možnosť vylúčiť výpočetné uzly pri špecifikácii požiadavkov úlohy pomocou definovania prefixu a suffixu parametru `worker_name`,
- automatické odpojenie uzlov pri ich dlhšej odmlke (v prípade použitia pravidelných správ typu ping),
- kontrola zaťaženia výpočetných uzlov a dynamické prihlasovanie/odhlasovanie z clustru,
- v prípade záujmu používateľov clustru, klientskú aplikáciu poskytujúcu "command line interface".

# Literatúra


[1] Docker security. Máj 2018, [cit. 14.5.2018].
URL https://docs.docker.com/engine/security/security/

[2] Repozitár zbernice projektu Testos. 2018, [cit. 14.5.2018].
URL https://pajda.fit.vutbr.cz/testos/testos-bus

[3] Runtime privilege and Linux capabilities. Máj 2018, [cit. 14.5.2018].
URL https://docs.docker.com/engine/reference/run/

[4] Bertrand Bordage: Avoid docker-py | Bertrand Bordage. Júl 2014, [cit. 14.5.2018].
URL http://blog.bordage.pro/avoid-docker-py/

[5] Buyya, R.; Yeo, C. S.; Venugopal, S.; aj.: Cloud computing and emerging IT platforms: Vision, hype, and reality for delivering computing as the 5th utility. *Future Generation Computer Systems*, ročník 25, č. 6, 2009: s. 599 – 616, ISSN 0167-739X, doi:https://doi.org/10.1016/j.future.2008.12.001.
URL http://www.sciencedirect.com/science/article/pii/S0167739X08001957

[6] Foster, I.; Kesselman, C. (editori): *The Grid: Blueprint for a New Computing Infrastructure*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1999, ISBN 1-55860-475-8.

[7] Gandotra, I.; Abrol, P.; Gupta, P.: Cloud computing over cluster, grid computing: a comparative analysis. *Journal of Grid and Distributed computing*, ročník 1, č. 1, 2011: s. 1–4.

[8] Gentzsch, W.: Sun Grid Engine: towards creating a compute power grid. In *Proceedings First IEEE/ACM International Symposium on Cluster Computing and the Grid*, 2001, s. 35–36, doi:10.1109/CCGRID.2001.923173.

[9] Janoušek, M.: *Dynamické analyzátory pro platformu SearchBestie*. Diplomová práca, Vysoké učení technické v Brně, Fakulta informačních technologií, Brno, CZ, 2017.
URL http://www.fit.vutbr.cz/study/DP/DP.php?id=19444

[10] Kaur, K.; Rai, A. K.: A Comparative Analysis: Grid, Cluster and Cloud Computing. 2014.

[11] Malík, V.: *Dynamická analýza použití knihovních volání*. Bakalářská práce, Vysoké učení technické v Brně, Fakulta informačních technologií, Brno, CZ, 2014.
URL http://www.fit.vutbr.cz/study/DP/BP.php?id=15992

[12] Min Ding, W.; Ghansah, B.; Yan Wu, Y.: Research on the Virtualization Technology in Cloud Computing Environment. *International Journal of Engineering Research in Africa*, ročník 21, December 2015: s. 191–196, doi:10.4028/www.scientific.net/JERA.21.191.

[13] Mouat, A.: *Using Docker*. O'Reilly, 2015, ISBN 978-1-4919-1576-9.
URL https://books.google.cz/books?id=zw2zrQEACAAJ

[14] Országh, M.: *Využití Linuxových kontejnerů pro stálost testů*. Bakalářská práce, Vysoké učení technické v Brně, Fakulta informačních technologií, Brno, CZ, 2017.
URL http://www.fit.vutbr.cz/study/DP/BP.php?id=19240

[15] Ryan Olson; Jonathan Calmels; Felix Abecassis; aj.: NVIDIA Docker: GPU Server Application Deployment Made Easy. Jún 2016, [cit. 14.5.2018].
URL https://devblogs.nvidia.com/nvidia-docker-gpu-server-application-deployment-made-easy/

[16] Skupina Testos: Domovská stránka projektu Testos [online]. FIT VUT v Brně. 2017, [cit. 14.5.2018].
URL http://testos.org

# Príloha A

# Typy správ posielaných po zbernici Testos

V tejto prílohe sú vymenované všetky typy správ, ktoré cluster posiela po zbernici Testos, spolu s ich popisom a definíciou tela správy. Toto telo je vo všetkých prípadoch kódované vo formáte json a správy sú posielané v textovej podobe.

V nasledujúcich podkapitolách sú správy rozdelené podľa toho, medzi ktorými časťami clustru prebiehajú, pričom poradie účastníkov komunikácie určuje smer správy:

- Klient - server: podkapitola A.1.
- Server - výpočetný uzol: podkapitola A.3.
- Výpočetný uzol - server: podkapitola A.4.

## A.1 Klient - server

Správy posielané od klienta určené pre server.

### A.1.1 org.testos.tcluster.server.task.new

Typ správy: Request
Popis: zadanie novej úlohy clustru.

Vysvetlenie jednotlivých atribútov (kľúčov) v tele správy (príklad znázornený na obrázku A.1):

- platform – môže obsahovať: *docker*, *libvirt*, *nvidia-docker*,
- image – názov obrazu disku,
- script – skript určený na spustenie (mal by obsahovať *shebang* - tj. riadok na začiatku skriptu začínajúci znakmi mriežka a výkričník, nasledovanými názvom interpretu, ktorý bude použitý na interpretáciu skriptu),
- input_artifacts_tgz_b64 – archív vo formáte *tar.gz* načítaný ako binárny súbor a zakódovaný do formátu *base64*, obsahujúci vstupné artefakty,

- estimated_time – reťazec určujúci odhadovaný čas behu (formát je vysvetlený v podkapitole 4.3.2 pri metóde `estimate_time()`),
- timeout – reťazec určujúci maximálny čas behu (formát vysvetlený v podkapitole 4.3.2 pri metóde `set_timeout()`),
- scheduling_priority – číslo určujúce plánovaciu prioritu pre úlohu (akceptované hodnoty sú 1-5, vyššie číslo znamená vyššiu prioritu),
- task_parameters – pole reťazcov, ktoré budú použité ako parametre úlohy (vysvetlené v podkapitole 3.2.1),
- host – objekt obsahujúci dva kľúče:
  - mandatory_requirements: prázdny objekt alebo objekt obsahujúci dvojice *požiadavok*: *hodnota*, pričom požiadavkom môže byť akýkoľvek reťazec predstavujúci atribút výpočetného stroja alebo jeden z rezervovaných kľúčových slov (vymenované nižšie). Všetky požiadavky okrem rezervovaných sú porovnávané s atribútmi výpočetných uzlov na zhodnosť.
    Medzi rezervované kľúčové slová patrí:
    * `ram.min_amount`: pamäť >= hodnota (hodnota > 0),
    * `cpu.min_cores`: počet jadier procesora >= hodnota (hodnota > 0),
    * `net.min_download`: rýchlosť siete (sťahovanie) >= hodnota,
    * `net.min_upload`: rýchlosť siete (nahrávanie) >= hodnota,
    * `net.max_latency`: odozva siete <= hodnota,
    * `worker_name_prefix`: prefix názvu výpočetného stroja,
    * `worker_name_suffix`: suffix názvu výpočetného stroja.

    Ostatné požiadavky:
    * `gpu`: prítomnosť grafického procesora (hodnota typu boolean),
    * `gpu_is_highend`: vysokovýkonný grafický procesor (hodnota typu boolean),
    * `cpu_arch`: architektúra procesora,
    * akýkoľvek atribút výpočetného stroja (pri týchto atribútoch sú povolené iba hodnoty typu boolean).
  - priority: pole obsahujúce akékoľvek atribúty výpočetných strojov (výpočetné stroje budú pri plánovaní zostupne zoradené podľa týchto atribútov, pričom sa dbá na poradie, v ktorom sú atribúty v poli uvedené)
- tags – pole reťazcov/štítkov slúžiacich pre kategorizáciu a označovanie úloh.

```
{
    "platform": "docker",
    "image": "ubuntu:16.04",
    "script": "#!/bin/sh\necho \"testos\"\necho \"argument: '$1'\"",
    "input_artifacts_tgz_b64": "",
    "estimated_time": "5m",
    "timeout": "30m",
    "scheduling_priority": 2,
    "task_parameters": [
        "0",
        "1",
        "2"
    ],
    "host": {
        "mandatory_requirements": {
            "ram.min_amount": 1024,
            "cpu_arch": "x86_64",
            "net.max_latency": "50",
            "gpu": true,
            "worker_name_suffix": "vutbr.cz",
            "opencl_min_ver_1.5": true
        },
        "priority": [
            "gpu_is_highend"
        ]
    },
    "tags": [
        "wirec_tasks"
    ]
}
```


Obr. A.1: org.testos.tcluster.server.task.new

Typ správy: Response


```
{
    "error": null,
    "task_id": 1,
    "task_instances": [
        {
            "task_instance_id": 1,
            "parameter": "0"
        },
        {
            "task_instance_id": 2,
            "parameter": "1"
        },
        {
            "task_instance_id": 3,
            "parameter": "2"
        }
    ]
}
```


Obr. A.2: Odpoveď na org.testos.tcluster.server.task.new

### A.1.2 org.testos.tcluster.server.task.abort

Typ správy: Signal

```
{
    "task_id": 1
}
```

Obr. A.3: org.testos.tcluster.server.task.abort

### A.1.3 org.testos.tcluster.server.task.get_instance_ids_by_param

Typ správy: Request

```
{
    "task_id": 1,
    "parameter": "1"
}
```

Obr. A.4: org.testos.tcluster.server.task.get_instance_ids_by_param

Typ správy: Response

```
{
    "task_instances": [
        {
            "task_instance_id": 2
        }
    ]
}
```

Obr. A.5: Odpoveď na org.testos.tcluster.server.task.get_instance_ids_by_param

### A.1.4 org.testos.tcluster.server.task.get_info

Typ správy: Request

```
{
    "task_instance_id": 2
}
```

Obr. A.6: org.testos.tcluster.server.task.get_info

Typ správy: Response

```
{
    "status": "COMPLETE",
    "started_at": 1526405880,
    "finished_at": 1526405881,
    "worker_name": "unique_name_1",
    "task_instance_id": 2
}
```


Obr. A.7: Odpoveď na org.testos.tcluster.server.task.get_info

### A.1.5 org.testos.tcluster.server.task.get_return_code

Typ správy: Request


```
{
    "task_instance_id": 2
}
```


Obr. A.8: org.testos.tcluster.server.task.get_return_code

Typ správy: Response


```
{
    "return_code": 77,
    "task_instance_id": 2
}
```


Obr. A.9: Odpoveď na org.testos.tcluster.server.task.get_return_code

### A.1.6 org.testos.tcluster.server.task.get_output

Typ správy: Request


```
{
    "task_instance_id": 2
}
```


Obr. A.10: org.testos.tcluster.server.task.get_output

Typ správy: Response


```
{
    "stdout_b64": "YmVmb3JlX3NsZWVwCmFyZ3VtZW50OiAnMCcKYWZ0ZXJfc2xlZXAK",
    "stderr_b64": "ZXJyb3I6YmVmb3JlX3NsZWVwCg=="
    "task_instance_id": 2
}
```


Obr. A.11: Odpoveď na org.testos.tcluster.server.task.get_output

### A.1.7 org.testos.tcluster.server.task.get_output_artifacts

Typ správy: Request

```
{
    "task_instance_id": 2
}
```

Obr. A.12: org.testos.tcluster.server.task.get_output_artifacts

Typ správy: Response

```
{
    "output_artifacts_tgz_b64": "",
    "task_instance_id": 2
}
```

Obr. A.13: Odpoveď na org.testos.tcluster.server.task.get_output_artifacts

## A.2 Server - klient

### A.2.1 org.testos.tcluster.task.finished

Typ správy: Signal

```
{
    "task_id": 1,
    "task_instance_id": 2
}
```

Obr. A.14: org.testos.tcluster.task.finished

## A.3 Server - výpočetný uzol

### A.3.1 org.testos.tcluster.worker.WORKER_ID.task.exec

Typ správy: Request

```
{
    "created_at": 1526411843,
    "platform": "docker",
    "image": "ubuntu:16.04",
    "script": "#!/bin/sh\necho \"testos\"\necho \"argument: '$1'\"",
    "estimated_time": 300,
    "timeout": 1800,
    "priority": 2,
    "host_priorities_json": "[\"gpu_is_highend\"]",
    "input_artifacts_path": null,
    "task_json": null,
    "status": "NEW",
    "parameter": "1",
    "task_instance_id": 2,
    "task_id": 1,
    "host_priorities": [
        "gpu_is_highend"
    ],
    "tags": [
        "tag_xy",
        "wirec_tasks"
    ],
    "host_requirements": {
        "ram.min_amount": "1024",
        "cpu_arch": "x86_64",
        "cpu.min_cores": "2",
        "net.min_download": "2",
        "net.min_upload": "1",
        "net.max_latency": "50",
        "gpu": true,
        "worker_name_prefix": "PC_XYZ",
        "worker_name_suffix": "vutbr.cz",
        "opencl_min_ver_1.5": "true"
    },
    "input_artifacts_tgz_b64": null
}
```


Obr. A.15: org.testos.tcluster.worker.WORKER_ID.task.exec

Typ správy: Response


```
{
    "success": true,
    "task_instance_id": 2
}
```


Obr. A.16: Odpoveď na org.testos.tcluster.worker.WORKER_ID.task.exec

### A.3.2 org.testos.tcluster.worker.WORKER_ID.task.abort

Typ správy: Request

```
{
    "task_instance_id": 2
}
```

Obr. A.17: org.testos.tcluster.worker.WORKER_ID.task.abort

Typ správy: Response

```
{
    "success": true,
    "task_instance_id": 2
}
```

Obr. A.18: Odpoveď na org.testos.tcluster.worker.WORKER_ID.task.abort

## A.4 Výpočetný uzol - server

### A.4.1 org.testos.tcluster.server.join

Typ správy: Request

```
{
    "worker_name": "unique_name_1",
    "docker_support": true,
    "nvidia_docker_support": false,
    "libvirt_support": false,
    "ram_amount": 4096,
    "cpu_cores": 4,
    "gpu": true,
    "gpu_is_highend": true,
    "cpu_arch": "x86_64",
    "net_download": 50,
    "net_upload": 20,
    "net_latency": 20,
    "custom_attributes": {
        "opencl_min_ver_1.5": true,
        "xy_support": true
    }
}
```

Obr. A.19: org.testos.tcluster.server.join

Typ správy: Response

```
{
    "error": "",
    "worker_id": 42
}
```

Obr. A.20: Odpoveď na org.testos.tcluster.server.join

### A.4.2 org.testos.tcluster.server.retire

Typ správy: Request

```
{
    "worker_id": 42
}
```

Obr. A.21: org.testos.tcluster.server.retire

Typ správy: Response

```
{
    "success": true
}
```

Obr. A.22: Odpoveď na org.testos.tcluster.server.retire

### A.4.3 org.testos.tcluster.server.task.started

Typ správy: Signal

```
{
    "task_instance_id": 2,
    "started_at": 1526405744
}
```

Obr. A.23: org.testos.tcluster.server.task.started

### A.4.4 org.testos.tcluster.server.task.finished

Typ správy: Request

```
{
    "task_instance_id": 2,
    "status": "COMPLETE",
    "started_at": 1526405880,
    "finished_at": 1526405881,
    "output_artifacts_tgz_b64": "",
    "return_code": 77,
    "stdout_b64": "YmVmb3JlX3NsZWVwCmFyZ3VtZW50OiAnMCcKYWZ0ZXJfc2xlZXAK",
    "stderr_b64": "ZXJyb3I6YmVmb3JlX3NsZWVwCg=="
}
```

Obr. A.24: org.testos.tcluster.server.task.finished

Typ správy: Response

```
{
    "received": true
}
```

Obr. A.25: Odpoveď na org.testos.tcluster.server.task.finished

# Príloha B

# Konfiguračné súbory

Oba konfiguračné súbory sú vo formáte `ini` a spracovávané sú pomocou modulu `configparser`[1] (modul zo štandardnej knižnice jazyka Python 3).

## B.1 Server

Príklad konfiguračného súboru serverovej aplikácie clustru:

```
1  [testos-bus]
2  default_cluster_channel = org.testos.tcluster
3
4
5  [database]
6  # either absolute path or path relative to the project root
7  db_file = runtime/server/database/tcluster.db
8
9
10 [server]
11 # ERROR / WARNING / INFO / DEBUG
12 log_level = DEBUG
13 # either absolute path or path relative to the project root
14 artifacts_dir = runtime/server/task_artifacts/
```

Obr. B.1: Súbor server.ini.sample

[1] https://docs.python.org/3/library/configparser.html

## B.2 Výpočetný uzol

Príklad konfiguračného súboru aplikácie Worker:

```
# BOOLEAN attributes may only contain values: yes/no, true/false, 1/0

[testos-bus]
# (required)
default_cluster_channel = org.testos.tcluster


[general]
# either absolute path or path relative to the project root
runtime_dir = runtime/worker/
# ERROR / WARNING / INFO / DEBUG
log_level = DEBUG


[reservable-resources]
# [MB] (required)
ram_amount = 4096
# number (required)
cpu_cores = 4
# boolean (required)
gpu = 1


[attributes]
# boolean (required)
docker_support = yes
# boolean (required)
nvidia_docker_support = no
# boolean (required)
libvirt_support = no
# boolean
gpu_is_highend = yes
# string
cpu_arch = x86_64
# mbps
net_download = 50
net_upload = 20
# ms
net_latency = 20
# string (required, unique in tcluster)
worker_name = my_favorite_worker.fit.vutbr.cz


[custom-attributes]
# this section can be extended by any boolean attribute task might require
opencl_min_ver_1.5 = yes
xy_support = yes
```

Obr. B.2: Súbor worker.ini.sample